impede allowance. Kindly charge any additional fee, or credit any surplus, to Deposit Account 50-0324, Order No. 30585/31.

Respectfully submitted,

SHEARMAN & STERLING

Anthony L. Meola
Reg. No 44936

Dated: March 30, 2001

By: _____
David E. Boundy
Registration No. 36,461

Mailing Address:
SHEARMAN & STERLING
599 Lexington Avenue
New York, New York 10022
(212) 848-4000
(212) 848-7179 Telecopier

COMPLEX INSTRUCTION SET COMPUTER

BACKGROUND

This application claims priority, as a continuation of U.S. application Serial No. 09/626,325, filed July 26, 2000, which is a continuation-in-part (C-I-P) of International Application Serial No. PCT/US00/02239, filed January 28, 2000, which is a continuation-in-part (C-I-P) of U.S. Provisional Application Serial No. 60/176,610, filed January 18, 2000, which are incorporated herein by reference.

The invention relates to implementation of a computer central processor.

Each instruction for execution by a computer is represented as a binary number stored in the computer's memory. Each different architecture of computer represents instructions differently. For instance, when a given instruction, a given binary number, is executed by an IBM System/360 computer, an IBM System/38, an IBM AS/400, an IBM PC, and an IBM PowerPC, the five computers will typically perform five completely different operations, even though all five are manufactured by the same company. This correspondence between the binary representation of a computer's instructions and the actions taken by the computer in response is called the Instruction Set Architecture (ISA).

A program coded in the binary ISA for a particular computer family is often called simply "a binary." Commercial software is typically distributed in binary form. The incompatibility noted in the previous paragraph means that programs distributed in binary form for one architecture generally do not run on computers of another. Accordingly, computer users are extremely reluctant to change from one architecture to another, and computer manufacturers are narrowly constrained in modifying their computer architectures.

A computer most naturally executes programs coded in its native ISA, the ISA of the architectural family for which the computer is a member. Several methods are known for executing binaries originally coded for computers of another, non-native, ISA. In hardware emulation, the computer has hardware specifically directed to executing the non-native instructions. Emulation is typically controlled by a mode bit, an electronic switch: when a non-native binary is to be executed, a special instruction in the emulating computer sets the mode bit and transfers control to the non-native binary. When the non-native program exits, the mode bit is reset to specify that subsequent instructions are to be interpreted by the native ISA. Typically, in an emulator, native and non-native instructions are stored in different address spaces. A

address space of a memory of the computer. The address translation circuit ~~for translating~~ TRANSLATES address references ~~is~~ generated by the program from the program's logical address space to the computer's physical address space. The profile circuitry is cooperatively interconnected with the instruction pipeline and is configured to detect, without compiler assistance for execution profiling, occurrence of profileable events occurring in the instruction pipeline, and cooperatively interconnected with the memory access unit to record profile information describing physical memory addresses referenced during an execution interval of the program.

Embodiments of the invention may include one or more of the following features. The recorded physical memory references may include addresses of binary instructions referenced by an instruction pointer, and at least one of the recorded instruction references may record the event of a sequential execution flow across a page boundary in the address space. The recorded execution flow across a page boundary may occur within a single instruction. The recorded execution flow across a page boundary may occur between two instructions that are sequentially adjacent in the logical address space. At least one of the recorded instruction references may be a divergence of control flow consequent to an external interrupt. At least one of the recorded instruction references may indicate the address of the last byte of an instruction executed by the computer during the profiled execution interval. The recorded profile information may record a processor mode that determines the meaning of binary instructions of the computer. The recorded profile information may record a data-dependent change to a full/empty mask for registers of the computer. The instruction pipeline may be configured to execute instructions of two instruction sets, a native instruction set providing access to substantially all of the resources of the computer, and a non-native instruction set providing access to a subset of the resources of the computer. The instruction pipeline and profile circuitry may be further configured to effect recording of profile information describing an interval of the execution of an operating system coded in the non-native instruction set.

In general, in a fifteenth aspect, the invention features a method. A program is executed on a computer. Profile information is recorded concerning the execution of the program, the profile information recording of the address of the last byte of at least one instruction executed by the computer during a profiled interval of the execution.

In general, in a sixteenth aspect, the invention features a method. A program is executed on a computer, without the program having been compiled for profiled execution, the program being coded in an instruction set in which an interpretation of an instruction depends on a

XP bit **184, 186** behaves somewhat analogously to a MESI (Modified, Exclusive, Shared, Invalid) cache protocol. The XP "unprotected" state is roughly equivalent to the MESI "Exclusive" state, and means that no information from this page may be cached while the page remains unprotected. The "protected" XP state is roughly equivalent to the MESI "Shared" state, and means that information from the page may be cached, but cached information must be purged before the page can be written. Four points of the analogy are explained in Table 2.

Table 2

MESI			
		fetch for sharing	write
Shared	cached		action 1
Exclusive	uncached / exclusive	action 2	3

TAXi XP protection			
		fetch for sharing	write
Protected			action 1
Unprotected	uncached / exclusive	action 2	3

action 1: discard all cached copies of the data, transition to the uncached/exclusive state
action 2: fetch a shared/duplicate copy, and transition to the cached/shared state.

A write to a MESI "Shared" cache line forces all other processors to purge the cache line, and the line is set to "Exclusive." Analogously, a write to an XP-protected **184, 186** page causes the page to be set to unprotected. These two analogous actions are designated "action 1" in Table 2. If a page's ISA bit **180, 182** is One and XP bit **184, 186** is One, then this is an X86 instruction page that is protected. Any store to an X86 ISA page whose XP bit **184, 186** is One (protected), whether the current code is X86 native code or TAXi code, is aborted and control is passed to the protected exception handler. The handler marks the page unprotected by setting the page's XP bit **184, 186** to Zero. Any TAXi code associated with the page is discarded, and PIPM database **602** that tracks the TAXi code is cleaned up to reflect that discarding. Then the store is retried – it will now succeed, because the page's XP bit **184, 186** has been cleared to Zero (unprotected). If TAXi code writes onto the X86 page of which this TAXi code is the translation, then the general mechanism still works – the exception handler invalidates the TAXi code that was running, and will return to the converter and original X86 text instead of the TAXi code that executed the store.

A write to a "Exclusive" cache line, or to an XP-unprotected **184, 186** page, induces no state change. If XP bit **184, 186** is Zero (unprotected), then stores are allowed to complete. These two states are labeled "3" in Table 2.

A read from a MESI "Shared" cache line proceeds without further delay, because the data in the cache are current. Analogously, converter **136** execution of an instruction from an XP-

protected **184**, **186** page proceeds without delay, because if any translated TAXi code has been generated from the instructions on the page, the TAXi code is current, and the profiling and probing mechanisms (**400**, **600**, see sections V and VI, *infra*) will behave correctly. These analogous responses are labeled "4" in Table 2.

A read from a cache line, where that cache line is held in another processor in "Exclusive" state, forces the cache line to be stored to memory from that other processor, and then the line is read into the cache of the reading processor in "Shared" state. Analogously, when converter **136** executes code from XP-unprotected **184**, **186** page (ISA is One, representing X86 code, and XP bit **184, 186** is Zero, indicating unprotected), and is about to write a profile trace-packet entry, with certain additional conditions, the machine takes an "unprotected" exception and vectors to the corresponding handler. The handler makes the page protected and synchronizes that page with other processors. These analogous actions are labeled "action 2" in Table 2. An unprotected exception is raised when an instruction is fetched from an unprotected X86 page (the page's I-TLB.ISA bit **182** is One, see section II, *infra*, and I-TLB.XP **186** bit is Zero), and TAXi_Control.unpr **468** is One and either of the following:

(1) a profile capture instruction is issued to start a new profile packet (TAXi_State.Profile_Active (**482** of **Fig. 4h**) is Zero, TAXi_State.Profile_Request **484** is One, and TAXi_State.Event_Code_Latch **486**, **487** contains an event code for which "initiate packet" **418** is True in **Fig. 4b**), or

(2) when the first instruction in a converter recipe is issued and TAXi_State.Profile_Active **482** is One.

The TAXi_State terms of this equation are explained in sections V.E and V.F and **Figs. 4g**, **4h**, **5a** and **5b**.

The unprotected exception handler looks up the physical page address of the fetched instruction from the EPC.EIP (the EPC is the native exception word (instruction pointer and PSW) pushed onto the stack by the exception, and EPC.EIP is the instruction pointer value), or from a TLB fault address processor register. The interrupt service routine sets the PFAT.XP bit **184** and I-TLB.XP bit **186** for the page to One, indicating that the page is protected. This information is propagated to the other Tapestry processors and DMU (DMA monitoring unit) **700**, in a manner similar to a "TLB shoot-down" in a shared-memory multiprocessor cache system. The exception handler may either abort the current profile packet (see section V.F, *infra*), or may put the machine in a context from which the profile packet can be continued. Then the exception handler returns to converter **136** to resume execution.

point **318**, or takes the Tapestry short path through GENERAL entry point **317**. (Note that routine **308** will have to be separately mapped **397** into the address space of Tapestry process **314** – recall that Tapestry process **314** is under the management of Tapestry OS **312**, while the address space **380** of an X86 process is entirely managed by X86 operating system **306**, entirely outside the ken of Tapestry operating system **312**.) If the same external interrupt **388** occurs (arrow (21)), the interrupt can be handled in Tapestry operating system **312** (outside the code of **Fig. 3j**), and control will directly resume (arrow (22)) at the instruction following the interrupt, without tracing through the succession of handlers. When Tapestry library routine **308** completes, control will return to the caller (arrow (23)) in the conventional manner. The only overhead is a single instruction **393**, setting the value of XD in case the callee is in X86 code.

H. Alternative embodiments

In an alternative embodiment, a "restore target page" of memory is reserved in the operating system region of the X86 address space. In PFAT **172**, ISA bit **180** for the restore target page is set to indicate that the instructions on the page are to be interpreted under the Tapestry instruction set. This restore target page is made nonpageable. At step **363** of **Fig. 3j**, the ~~EIP~~ EPC.EIP value is replaced with an X86 address pointing into the restore target page, typically with byte offset bits of this replacement ~~EIP~~ EPC.EIP storing the number of the save slot. In an alternative embodiment, the ~~EIP~~ EPC.EIP is set to point to the restore target page, and the save slot number is stored in one of the X86 registers, for instance EAX. In either case, when X86 operating system **306** resumes the thread, the first instruction fetch will trigger an X86-to-Tapestry transition exception, before the first actual instruction from the restore target page is actually executed, because the restore target page has the Tapestry ISA bit set in its PFAT and I-TLB entries. X86-to-Tapestry transition handler **320** begins by testing the address of the fetched instruction. An address on the restore target page signals that there is extended context to restore. The save slot number is extracted from the instruction address (recall that the save slot number was coded into the EPC or EAX on exception entry, both of which will have been restored by X86 operating system **306** in the process of resuming the thread). The processor context is restored from the save slot, including the EPC.EIP value at which the thread was originally interrupted. In an alternative embodiment, only the extended context (not including the X86 context) is restored from the save slot, so that any alterations to the X86 context effected by Tapestry operating

virtual address space, when two successive instructions are separated by a physical page boundary, or when a single instruction straddles a virtual page boundary. (As is known in the art, two pages that are sequential in a virtual address space may be stored far from each other in physical memory.) By managing the X86 pages in the physical address space, Tapestry operates at the level of the X86 hardware being emulated. Thus, the interfaces between Tapestry and X86 operating system **306** may be as well-defined and stable as the X86 architecture itself. This obviates any need to emulate or account for any policies or features managed by ~~the~~ X86 operating system **306**. For instance, Tapestry can run any X86 operating system (any version of Microsoft Windows, Microsoft NT, or IBM OS/2, or any other operating system) without the need to account for different virtual memory policies, process or thread management, or mappings between logical and physical resources, and without any need to modify X86 operating system **306**. Second, if X86 operating system **306** shares the same physical page among multiple X86 processes, even if at different virtual pages, the page will be automatically shared. There will be a single page. Third, this has the advantage that pages freed deleted from an address space, and then remapped before being reclaimed and allocated to another use,

Referring to **Fig. 4b**, events are classified into a fairly fine taxonomy of about thirty classes. Events that may be recorded include jumps, subprogram CALL's and returns, interrupts, exceptions, traps into the kernel, changes to processor state that alters instruction interpretation, and sequential flow that crosses a page boundary. Forward and backward jumps, conditional and unconditional jumps, and near and far jumps are distinguished.

Referring to **Figs. 4g** and **4h**, profiler **400** has a number of features that allow profiling to be precisely controlled, so that the overhead of profiling can be limited to only those execution modes for which profile analysis is desired.

Referring to **Figs. 5a** and **5b**, as each X86 instruction is decoded by the converter (**136** of **Fig. 1c**), a profile entry is built up in a 64-bit processor register **594**. During execution of the instruction, register **594** may be modified and overwritten, particularly if the instruction traps into Tapestry operating system **312**. At the completion of the instruction, profiler **400** may choose to capture the contents of the profile entry processor register into a general register.

Hot spot detector **122** recognizes addresses that frequently recur in a set of profile packets. Once a hot spot is recognized, the surrounding entries in the profile may indicate (by physical address) a region of code that is frequently executed in correlation with the recurring

processor PSW. A successful probe leaves the Probe_Mask **620** unaltered. Thus, classes of probeable events remain enabled as long as each probe in the class is successful.

By resetting the EPC.EIP to point to TAXi translated code, the RFE instruction at the end of the probe exception handler effects a transition to the TAXi code. Because the TAXi code was transliterated from X86 code, it follows the X86 convention, and thus the argument copying that would have been performed by the transition exception handler (see sections II, III, and IV, *supra*) is not required. Further, because both the probe exception handler and the TAXi code are in Tapestry ISA, no probe exception occurs on this final transition.

When a probe exception is triggered, and the software probe fails to find a translation, several steps are taken. The bit in Probe_Mask **620** that corresponds to the event that triggered the probe is cleared to Zero, to disable probes on this class of event until the next expiry of Probe_Timer **630**. This is accomplished by the Probe_Failed RFE signal and the remembered Decoded_Probe_Event latch **680**. The interrupt service routine returns using an RFE with one of two special "probe failed" event codes of **Fig. 4b**. Event code 0.0011 forces a reload of TAXi_Timers.Probe_Timer **630** with the Probe_Timer_Reload_Constant **632**. Event code 0.0010 has no side-effect on Probe_Timer **630**. It is anticipated that when a probe on a backwards branch fails, Probe_Timer **630** should be reset, by returning from the probe exception with an RFE of event code 0.0011, in order to allow the loop to execute for the full timer value, with no further probe exceptions. On the other hand, it is anticipated that when a probe on a "near CALL" fails, testing other near calls from the same page should be allowed as soon as Probe_Timer **630** expires, and thus this probe exception will return with an event code of 0.0010. The RFE returns to the point of the probe exception, and execution resumes in converter **136**.

If an RFE instruction that modifies Probe_Mask **620** is executed at the same time that the probe timer expiry attempts to reset Probe_Mask **620**, then the RFE action has higher priority and the reset request is discarded.

E. Additional features of probing

In the intermediate step **690** mentioned briefly *supra*, a bit vector of bits indicates whether a translation exists for code ranges somewhat finer than the page level encoded in the PFAT probe bits. After a probeable event occurs, and the class of that event is screened against the PFAT probe bits and the probe mask, the hardware tests the bit vector (in an operation